Using the OS X Keychain to store and retrieve passwords

Archived: 2026-04-05 22:42:00 UTC

November 5th, 2015

This document describes how to use the Mac OS X Keychain to store and retrieve passwords or other secrets from the command-line. This may be a reasonable solution for you to manage per-user secrets on your laptop or desktop, for example. Please consult with your friendly neighborhood paranoid.

Please see this page for a related discussion around passing passwords to different tools.

Let's assume that a 'playground' service requires a password to be available in your shell's `PLAYGROUND` environment variable. Placing the password into your `~/.bashrc` would accomplish this:

```
export PLAYGROUND="monkeybars"
```

Unfortunately, even with suitable permissions on `~/.bashrc`, this is not a good solution, since dot files like this have a tendency to accidentally be committed to `github.com` or copied to other places such that even though it might be reasonable to store such a password on your laptop in a plain text file protected by Unix permissions (e.g. `0400`), doing so would lead to an increased risk of exposure.

Instead, we'd like a method to retrieve the password non-interactively from a safe password store, thus only making it available in the environment, but not in a file.

A note about password managers

For common per-user passwords for use with HTTPS, such as used to log into common web services, please do use a password manager. Your password manager may also be suitable to store any generic secrets.

However, in some cases you may have a need to access such secrets on the command-line, and many password managers do not have trivial or convenient access tools to accomplish this.

In those cases, using the approach outlined here *may* be preferable.

Advantages of using the keychain

Using the OS X keychain means that you have easily accessible and usable encrypted storage of your secret. When a keychain is unlocked (such as most commonly the default "login" keychain at, well, login time), you can get the values of the items stored in it via the command-line or its GUI application.

This means you do not have to worry about remembering which file you may have stored a secret in, if the file is suitably protected using permissions or can be accessed by other users on the system.

Disadvantages of using the keychain

Using the OS X keychain also means that anybody with access to the keychain may have access to the secret. Most notably, your default login keychain is often unlocked when you're using your OS X system, so any process running with your privileges (or the super-user privileges, of course) may be able to access the secrets.

Individually encrypted storage such as by way of a password manager may be preferable, depending on your use case.

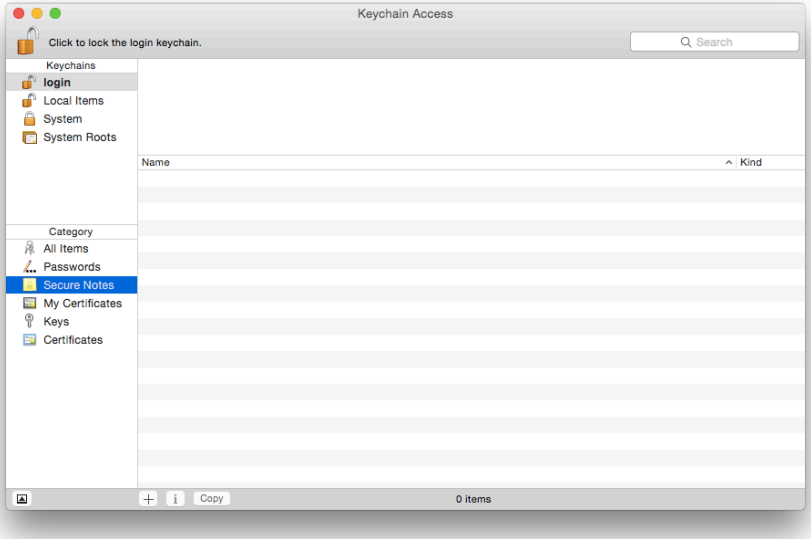
Adding a secret to your keychain

To add a secret to your keychain, you can either use the graphical application ("Keychain Access") or the command-line utility `security(1)`.

In the following example, we will create a password for an application called "playground":

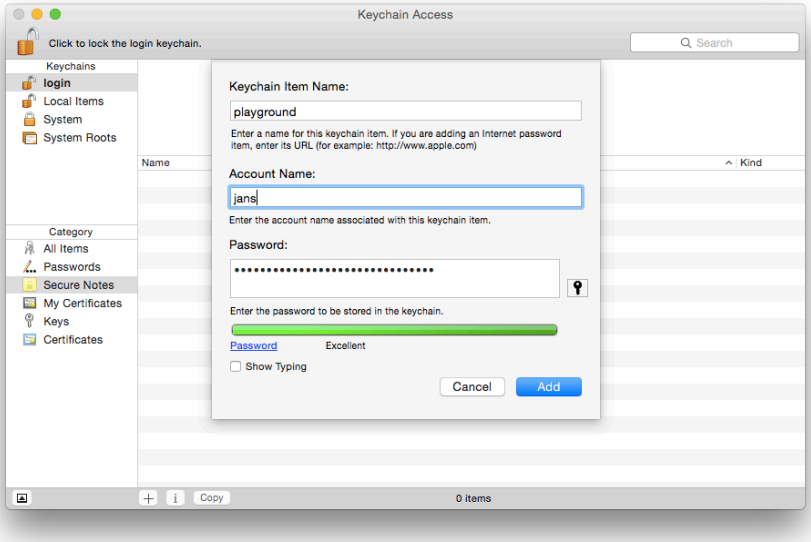
Keychain Access

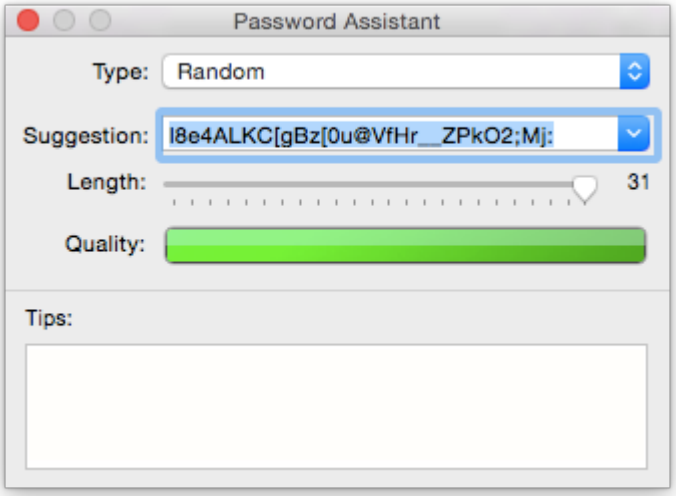
```
open /Applications/Utitlies/Keychain\ Access.app
```



Select `File->New Password Item` (cmd+N)

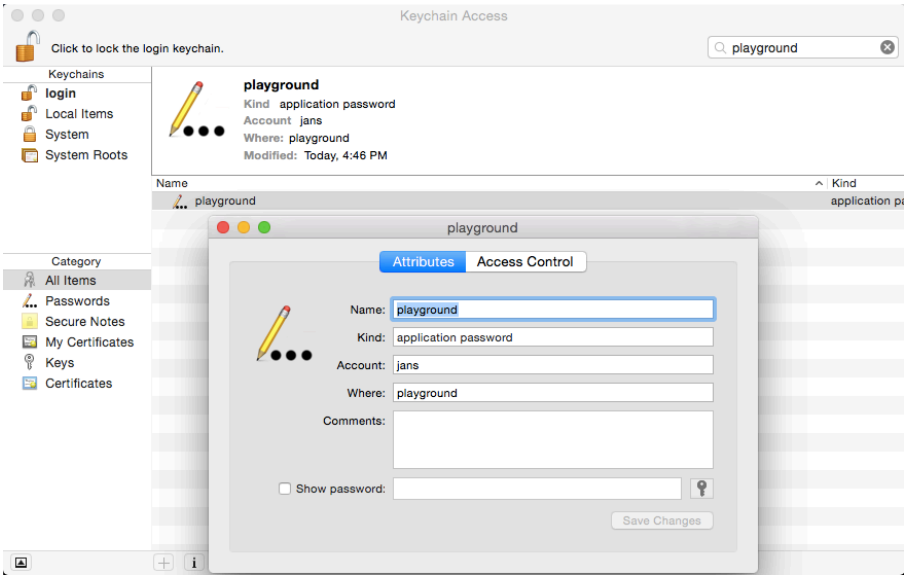
Use the Password Assistant to have it generate a new random password for you:





After clicking 'Add', your new password is now available in the login keychain.

Select 'All Items' and search for 'playground' to find it:



Via the command-line

First, we need to generate a new password. Don't try to be clever and think of one yourself -- no matter what you're friends tell you, you're not very random.

Let's create a quick little script to generate a reasonably random complex password:

```
$ cat > ~/bin/pwgen <<"EOF"
#! /bin/sh
num=32
if [ x"${1}" = x"-n" ]; then
  num=$2
fi
LANG=C tr -dc '[:print:]' </dev/urandom | head -c ${num}
EOF
$ chmod 755 ~/bin/pwgen
$
```

(Note: you could add a trailing "`| xargs`" to get a trailing `\n`, but then you'd need to exclude a few characters from the character set to avoid errors due to unterminated quotes etc. Similarly, you may wish to exclude spaces from your password charset. In other words: season the args to `tr(1)` to your liking.)

Now that we have a command to spit out a reasonably random, complex password, we can proceed to add it our keychain using the `security(1)` command:

```
$ security add-generic-password -a ${USER} -s playground -w $(pwgen)
```

A few things to note here:

The command will fail if you already have a "service" by the name of "playground" such as if you followed the GUI example above. You can delete the previously created password in the keychain using the command:

```
$ security delete-generic-password -a ${USER} -s playground
```

Secondly, we are necessarily leaking the newly generated password into the process table as the shell expands the subshell (more details here). Sadly, the `security(1)` utility has no other method of non-interactively accepting a password but on the command-line[1]. For this reason, it may be preferable to create the secret using the GUI application.

Retrieving the password from the command-line

In order to retrieve the password from the command-line, you can run the `security(1)` command as follows:

```
$ security find-generic-password -a ${USER} -s playground -w
l8e4ALKC[gBz[0u@VfHr__ZPkO2;Mj:
$
```

Using this approach, we can then place a password into the shell environment without leaking it into a file:

```
$ cat >> ~/.bashrc <<"EOF"
export PLAYGROUND=$(security find-generic-password -a ${USER} -s playground -w)
EOF
$
```

Final notes

Managing secrets appropriately requires a full understanding of what the secret is used to protect, who has access to the resources in question, and the circumstances of using the tools involved. No one solution fits all needs, and an approach that works for some situations may not be suitable for others.

The use case described here is specifically intended for ease of use while retaining storage safety of the secrets, but does not protect against an attacker with super-user- or same-user-level privileges on the same system.

When in doubt, please consult with your friendly neighborhood paranoid.

November 5th, 2015

1:

The `security(1)` utility offers the `-i` flag to enter interactive mode, allowing you to provide the password without leaking it into the process table, but that does not allow shell expansion within the interactive prompt:

```
$ security -i
security> add-generic-password -a jschauma -s playground -w gVmMQgEEF[bc'S15_=zD*]sHQ[IkOVY_
```

```
security> ^D
$
```

Alternatively, you can also let `security(1)` prompt you interactively for the password, by leaving out the last argument to the '`-w`' flag:

```
$ security add-generic-password -a ${USER} -s playground -w
password data for new item:
retype password for new item:
$
```

Source: https://www.netmeister.org/blog/keychain-passwords.html